To appear: 13th ACM Symposium on Operating Systems Principles, 1991

SCHEDULING AND IPC MECHANISMS FOR CONTINUOUS MEDIA

Ramesh Govindan
David P. Anderson

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

Next-generation workstations will have hardware support for digital "continuous media" (CM) such as audio and video. CM applications handle data at high rates, with strict timing requirements, and often in small "chunks". If such applications are to run efficiently and predictably as user-level programs, an operating system must provide scheduling and IPC mechanisms that reflect these needs. We propose two such mechanisms: *split-level CPU scheduling* of lightweight processes in multiple address spaces, and *memory-mapped streams* for data movement between address spaces. These techniques reduce the the number of user/kernel interactions (system calls, signals, and preemptions). Compared with existing mechanisms, they can reduce scheduling and I/O overhead by a factor of 4 to 6.

1. INTRODUCTION

Support for digital audio and video as I/O media is an important direction of computer systems research. We call audio and video *continuous media* (CM) because they are perceived as continuous, in contrast with discrete media such as graphics. There are various ways to incorporate CM in computer systems; in the *integrated* approach, CM data (digital audio and compressed digital video) is handled by user-level programs on general purpose operating systems such as Unix or Mach.

On existing general purpose OSs, integrated CM applications can suffer from poor performance; ACME [4] is one such application. ACME is a user-level I/O server that provides shared, network-transparent access to devices such as video cameras, speakers, and microphones (see Figure 1). We have implemented a prototype of ACME for a Sun SPARCstation running SunOS 4.1. It suffers from timing errors and lost data when there is concurrent system activity, even though the hardware is easily able to handle the data rates (*e.g.*, 64 Kb/sec audio data). The server also cannot supply the low delay needed for a telephone conversation client.

These problems are partly due to the overhead of *user/kernel interaction mechanisms* by which user-level programs invoke system functions such as CPU scheduling and I/O. This overhead includes user/kernel *domain switches* and *mapping switches*

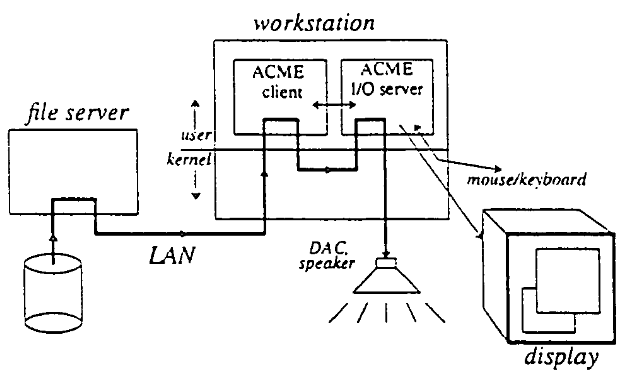

Figure 1: Audio playback is a basic integrated CM application. The client reads CM data from a file and sends it to the ACME server (bold line). The client also provides a graphical interface for making selections and controlling the playback parameters.

between different user virtual address spaces. For example, the UNIX asynchronous I/O mechanism requires up to ten domain switches and two mapping switches to read a block of data. The expense of these operations can be amortized by hysteresis and increased granularity (techniques used in buffered I/O and pipes). For CM applications, however, these techniques may increase delay excessively.

With the goal of better supporting integrated CM applications, we have designed OS mechanisms for scheduling and IPC.

- *Split-level scheduling and synchronization.* In this approach each user virtual address space (VAS) contains multiple lightweight processes (LWPs). The scheduler is partitioned into user-level and kernel-level parts, which communicate via shared memory. The information in shared memory is used to correctly prioritize LWPs in different VASs, and avoid domain and mapping switches where possible. Split-level scheduling can be used with many scheduling policies; we discuss its use for *deadline/workahead scheduling*, a real-time policy designed for CM.
- *Memory-mapped streams.* A memory mapped stream (MMS) is a shared-memory FIFO used for communicating CM data between user and kernel VASs. Once the MMS has been setup, no explicit kernel requests are needed to transfer data, and a minimal number of domain switches are needed for producer/consumer synchronization and I/O initiation.

In the next section we explain the process structure of the ACME server and the deadline/workahead scheduling policy in more detail. Sections 3 and 4 describe the new mechanisms. Section 5 gives some performance estimates, and Section 6 discusses related work.

2. PROCESS STRUCTURE AND SCHEDULING FOR CM APPLICATIONS

To motivate subsequent sections, we sketch a typical CM application (the ACME I/O server), and describe the deadline-workahead CPU scheduling policy.

2.1. The ACME Continuous Media I/O Server

ACME (Abstractions for Continuous Media) [4] supports applications such as audio/video conferencing, editing, and browsing. ACME allows its clients to create *logical devices*, associate them with physical I/O devices (video display or camera, audio speaker or microphone), and do I/O of CM data over *CM connections* (network connections carrying CM data). The data stream on a given CM connection may be multiplexed among different logical devices. ACME provides mechanisms for synchronizing different streams.

The ACME server performs multiple concurrent activities, and it is convenient to structure it as a set of concurrent processes. Our prototype uses the

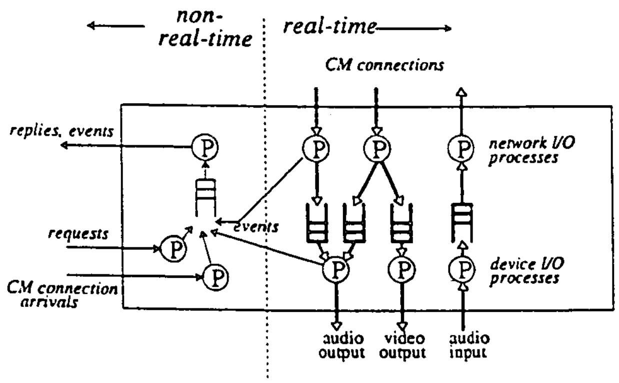

Figure 2: A CM application such as the ACME server consists of multiple processes sharing a single address space. Some of these processes handle streams of CM data, while others handle discrete events.

following processes (see Figure 2):

- For each CM connection, a *network I/O process* transfers data between an internal buffer and the network. It may do software processing (*e.g.*, volume scaling for audio streams).
- For each CM I/O device there is a *device I/O process*. For an output device, this process merges the data from the logical devices mapped to it and writes the resulting data to the device.
- *Event-handling processes* handle non-real-time events such as commands from the window server and requests for CM connection establishment.

The current implementation of ACME runs on the Sun SPARCstation. It is written in C++ and uses a preemptive lightweight process library. I/O is done using UNIX asynchronous I/O. The server handles telephone-quality (64 Kbps) audio I/O and video output, both compressed and uncompressed.

2.2. Deadline/Workahead Scheduling

The *Deadline/Workahead Scheduling* (DWS) CPU scheduling policy is designed for integrated CM [2]. In the DWS model, a process that handles CM data is called a *real-time process*. There are two classes of non-real-time processes: *interactive* (for which fast response time is important) and *background*.

A real-time process handles a sequence of *messages* each with a *logical arrival time* $l(m)$, either derived from a timestamp in the data or implicit from its position in the stream. Each real-time process has a fixed *logical delay bound*; the processing of each message should be finished within this amount after its logical arrival. At a given time t, a real-time process is called *critical* if it has an unprocessed message m with $l(m) \leq t$ (*i.e.*, *m*'s logical arrival time has passed). Real-time processes that have pending messages but are not critical are called *workahead* processes.

The DWS policy is as follows (see Figure 3). Critical processes have priority over all others, and are preemptively scheduled earliest deadline first (the deadline of a process is the logical arrival time of its first unprocessed message plus its delay bound). Interactive processes have priority over workahead processes, but are preempted when those processes become critical. Non-real-time processes are scheduled according to an unspecified policy, such as the UNIX time-slicing policy. The scheduling policy for workahead processes is also unspecified, and may be chosen to minimize context switching.

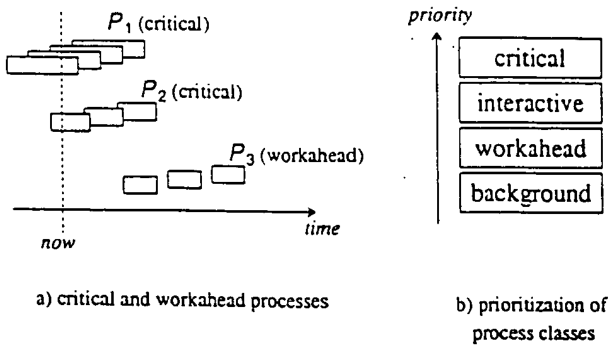

Figure 3: In the deadline/workahead scheduling (DWS) policy, each real-time process has a queue of pending messages. In example a), each message is shown as a rectangle whose left edge is its *logical arrival time* and whose right edge is its *deadline*. P_1 and P_2 are *critical* because they have a pending message whose logical arrival time is in the past. Processes are prioritized as shown in b). Critical processes are executed earliest deadline first; policies for other classes are unspecified.

3. SPLIT-LEVEL SCHEDULING AND SYNCHRONIZATION

CM applications are most easily programmed using multiple processes sharing a virtual address space (VAS). The two common multiprogramming techniques, *lightweight processes* (LWPs) and *threads*, each have advantages. LWPs are implemented purely at the user level, so context switches within a VAS is fast (on the order of tens of instructions). However, LWPs in different VASs may not be prioritized correctly. On the other hand, threads in different VASs can be correctly prioritized but context switches always involve an expensive user/kernel interaction.

Split-level scheduling is a scheduler implementation technique that combines the advantages of threads and LWPs: it minimizes user/kernel interactions while correctly prioritizing LWPs in different VASs. In the uniprocessor version of split-level scheduling[1], multiple LWPs per VAS share a single thread. An LWP sleeps or changes its priority by calling a *user-level scheduler* (ULS) (see Figure 4). The

[1] The technique is applicable to multiprocessor scheduling as well. For brevity we describe only the uniprocessor case.

ULS checks whether its VAS still contains the globally highest-priority LWP; this is done by examining an area of memory shared with the kernel. If so, the LWP context switch is done without kernel intervention. Otherwise, a kernel trap is done, and the *kernel-level scheduler* (KLS) decides which VAS should now execute, again based on information in shared memory.

While split-level scheduling can be used with many scheduling policies, we focus on its implementation for the deadline/workahead (DWS) policy described in the previous section. We also describe a related mechanism for efficient mutual exclusion between LWPs. For simplicity, we consider only the scheduling of real-time processes. It is straightforward to handle interactive and background processes as well (a VAS could contain a mixture of process types).

3.1. Client Interface to the Split-Level DWS Scheduler

A user-level library provides the client interface to the split-level DWS scheduler. The library exports interfaces for creating and destroying LWPs. An LWP P has three scheduling parameters: a fixed *delay bound* (see Section 2.2), a *critical time* C_P (the logical arrival time of its next message) and a *deadline* D_P (C_P plus the delay bound). The library provides the following functions for scheduling LWPs:

```
time_advance(TIME critical_time);
```

An LWP P calls this when it finishes a message; the argument is the logical arrival time of the next message. `time_advance()` updates C_P, and may yield the CPU.

```
timed_sleep(TIME critical_time);
```

An LWP calls this to suspend its execution until the given time; at this point it becomes runnable and C_P is set to the current time. This may be used by processes that do time-based output with no device synchronization (*e.g.*, slow video) or for rate-based flow control.

```
IO_wait(DESCRIPTOR iodesc,
    TIME critical_time);
```

An LWP calls this to wait for I/O to become possible on the given I/O descriptor representing a file, socket, I/O device or MMS (Section 4). When data arrives on the descriptor, the process becomes runnable and its C_P is set to the given value.

```
mask_LWP_preemption();
unmask_LWP_preemption();
```

These calls bracket "critical sections" within which the calling LWP cannot be preempted by an LWP in the same VAS.

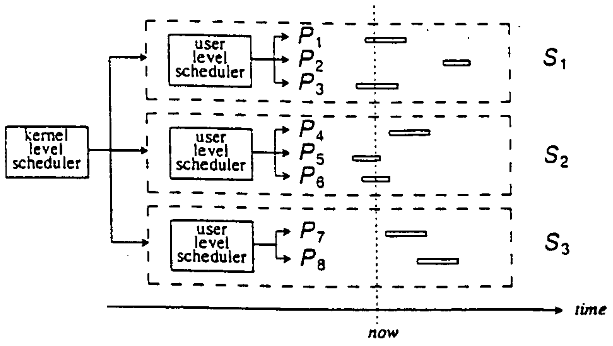

Figure 4: Using *split-level scheduling*, the kernel-level scheduler decides which user VAS should execute, and each VAS has a user-level scheduler (ULS) that manages the LWPs in that VAS. In this example, the KLS chooses VAS S_2 to run because it has the globally earliest deadline. The ULS in that VAS executes P_5, which has this deadline. User/kernel interactions can often be avoided: in this example, if P_5 yields then the context switch to P_6 (the next earliest deadline) can be done without a kernel call.

3.2. Implementation of the Split-Level DWS Scheduler

In this section we first describe the control and shared memory interfaces between the ULS and the KLS (see Figure 5). We then describe the implementation of each level. We defer discussing synchronization issues (*e.g.*, mutual exclusion on shared data structures) until Section 3.3.

3.2.1. User/Kernel Control Interface

The control interface between a ULS and the KLS consists of *system calls* and *user-interrupts*. The system call mechanism is the same as in UNIX-type systems: a trap instruction and return. The split-level scheduler needs one new system call: `yield()` yields the processor to another VAS.

User-interrupts are like UNIX signals except that the handler does not end with a system call to reset the signal mask (hence there is one domain switch rather than three). Each ULS registers the addresses of its handlers during initialization. Three types of user-interrupts are used: `INT_TIMER` is delivered when a timer elapses, `INT_IO_READY` is delivered when I/O becomes possible on an I/O descriptor, and `INT_RESUME` is delivered when a user VAS resumes after being preempted.

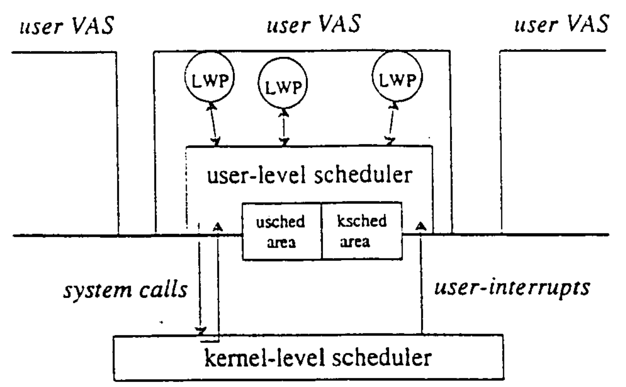

Figure 5: The user-level and kernel-level parts of the split-level scheduler communicate using *system calls*, *user-interrupts*, and through an area of shared memory.

3.2.2. User/Kernel Shared Memory Interface

The ULS for each VAS A shares a region of physical memory with the kernel. This region consists of two parts: the *usched* area and the *ksched* area (see Figure 5). The *usched* area is written by the ULS and read by the KLS. It contains the following:

D_A: the minimum of D_P for critical processes $P \in A$, or $+\infty$ if there are none. (A runnable LWP P is *critical* if $C_P < T_{now}$ and *workahead* if $C_P > T_{now}$). In other words, D_A is the earliest deadline of a critical LWP in A.

A *runnable* flag, TRUE if there are any runnable LWPs (critical or workahead) in the VAS.

A table of workahead and sleeping LWPs P in A such that $D_P < D_A$. Each entry in the table contains the critical time and deadline of the LWP.

For each I/O descriptor, a *waiting_for_IO* flag indicating whether an LWP is blocked on the descriptor, and if so the critical time and deadline of the LWP.

T_{next}: the time at which the next `INT_TIMER` user-interrupt should be delivered.

The ksched area, written by the KLS and read by the ULS, contains the following:

T_{now}: the current real time as measured by a hardware clock.

$D_{\bar{A}}$: the earliest deadline of a critical LWP not in A.

For each I/O descriptor, a *ready_for_IO* flag to indicate that data has arrived on that descriptor.

We use the following additional notation:

P_A: the highest priority runnable LWP in A. If D_A is finite then P_A is the earliest-deadline runnable critical LWP. Otherwise, P_A is set to an arbitrary runnable LWP (the choice of P_A in this case depends on the policy for workahead processes, which we do not specify).

P^*: the globally highest priority LWP.

A^*: the VAS containing P^*.

3.2.3. ULS Implementation

The ULS of VAS A is responsible for scheduling LWPs in A. If the ULS detects from its ksched area that $A \neq A^*$, it calls `yield()`. Similarly, if the KLS detects from A's usched area that $A \neq A^*$, it preempts A.

The ULS may need to preempt the currently running LWP when the critical time of a sleeping LWP is reached or a non-running workahead LWP becomes critical. This requires an `INT_TIMER` user-interrupt from the kernel. To reduce the number `INT_TIMER` user-interrupt deliveries, the following policy is used (see Figure 6):

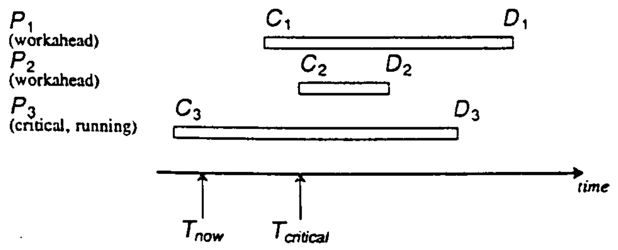

Figure 6: At a given time T_{now}, the ULS for a VAS A must have a pending `INT_TIMER` user-interrupt for the earliest critical time of a sleeping or workahead process $P \in A$ such that $D_P < D_A$. In this example, P_3 is critical and P_1 and P_2 are workahead. If P_3 is still running when C_{P_2} arrives, P_2 becomes critical and must preempt P_3. On the other hand, P_1 cannot preempt P_3 because its deadline is greater. Therefore a timer is needed for C_2 but not C_1.

Let X be the set of sleeping and workahead LWPs P in A such that $D_P < D_A$. Let $T_{critical} = \min(C_P: P \in X)$. Then it is sufficient for the ULS to maintain a timer for $T_{critical}$.

In addition to the data in the usched area, the ULS maintains queues of sleeping, critical and workahead LWPs. The implementations of `timed_sleep()`, `time_advance()`, and `IO_wait()` are as follows. Each function inserts the calling LWP into the appropriate structure (the sleep queue, the workahead or critical queue, and an I/O descriptor respectively), then does the following (see Figure 7):

(1) For each LWP P in the workahead and sleep queues such that $C_P < T_{now}$, insert P in the critical queue. For each LWP P sleeping on an I/O descriptor for which the *ready_for_IO* flag is set, insert P into the workahead or critical queues as appropriate.

(2) Update D_A in the usched area.

(3) Update the usched area's table of sleeping and workahead LWPs P with $D_P < D_A$ and the list of LWPs waiting for I/O.

(4) If $A \neq A^*$ then call `yield()`, else,

(5) Set $T_{next} = T_{critical}$. Do a context switch to P_A.

The handler for an `INT_TIMER` user-interrupt moves the LWPs for which $C_P \leq T_{now}$ from the sleep queue to the critical queue. It then executes steps 2, 3 and 5 above. The handler for an `INT_IO_READY` user-interrupt moves all LWPs for which the *ready_for_IO* flag is set to the critical or workahead queue and executes steps 1-5 above.

An `INT_RESUME` user-interrupt is delivered to a VAS when it resumes execution after having been preempted. Between when the VAS was preempted and T_{now}, an indeterminate amount of time has elapsed. The same is true when the VAS returns from a `yield()` system call. In both cases, the ULS performs steps 1-5 above to update its state.

3.2.4. KLS Implementation

The KLS is responsible for updating $D_{\bar{A}}$ in the ksched area of the currently executing VAS A. If in doing so it detects that $A \neq A^*$, it preempts A and switches to A^*.

Changes to $D_{\bar{A}}$ can occur when a sleeping LWP wakes up or a workahead LWP becomes critical. Timers have to be set for these moments. KLS timer management is analogous to that of a ULS. The KLS maintains a timer for the earliest C_P such that $D_P < D_{\bar{A}}$; this is computed from the tables in the usched areas of all VASs not currently executing. If, when the timer expires, the current VAS A is no longer A^*, the KLS preempts A and switches to the new A^*. Additionally, the kernel clock interrupt handler polls T_{next} in the usched area of A^*, delivering an `INT_TIMER` if necessary.

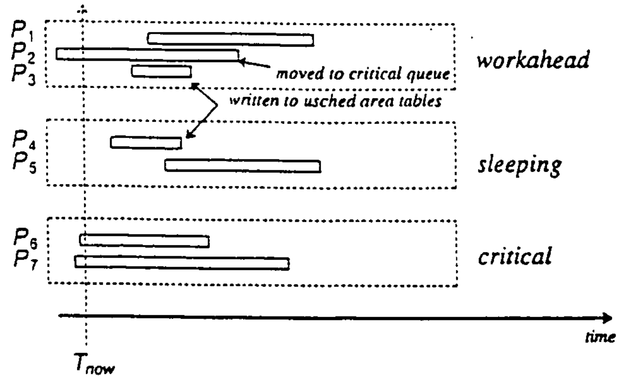

Figure 7: In this example, a VAS contains LWPs $P_1 \cdots P_7$. The current process, P_4, has called `timed_sleep()`, and the ULS has inserted it in the sleeping queue. The ULS then does the following (see Section 3.2.3): it moves P_2 to the critical queue, records P_3 and P_4 in the usched area, sets D_A to D_6, and sets a timer for C_4. Finally, it does a context switch to P_6.

The `yield()` system call determines A^*. It then computes $D_{\bar{A}}$, writes it to A^*'s ksched area and updates the pending timer if necessary. Finally, it switches to A^*, either by returning from an earlier `yield()` system call, or by delivering an `INT_RESUME` user-interrupt.

The handler for an I/O completion interrupt examines the *waiting_for_IO* flag for the corresponding descriptor. If it is set, the interrupt handler sets the *ready_for_IO* flag in the ksched area of the VAS A containing the descriptor. Moreover, if the LWP waiting on that descriptor is critical and has the earliest deadline, an `INT_IO_READY` user-interrupt is delivered, preempting the current VAS if necessary. If $A \neq A^*$, the handler updates D_A in A's ksched area, depending on whether the waiting LWP is critical.

3.3. Split-Level Synchronization

ULS/KLS shared memory can be concurrently accessed by multiple entities (LWPs, user-interrupt handlers and kernel interrupt handlers). We require mechanisms to synchronize access to this shared memory. By analyzing how specific shared data structures are accessed by different entities, we can obtain a set of specialized synchronization mechanisms that minimize user/kernel interactions.

First, ULS data structures such as the critical, workahead and sleeping queues are read and written by LWPs and user-interrupt handlers. To synchronize access to such structures it suffices to inhibit (or "mask") user-interrupts (since preemptive context switches within the VAS take place only in user-interrupt handlers, this inhibits LWP preemption as well). User-interrupt masking can also be used to implement `mask_LWP_preemption()` and `unmask_LWP_preemption()`, which provide mutual exclusion for client-defined data structures.

A technique called *virtual user-interrupt masking* provides user-interrupt masking without user/kernel interactions in the normal case. This technique uses a *mask level* in the usched area and a *request flag* in the ksched area. The request flag is a bitmap with one flag per user-interrupt type. To mask user-interrupts, the ULS increments the mask level. Whenever the kernel wants to deliver an interrupt and finds its mask level nonzero, it sets the corresponding bit in the request flag. When the ULS unmasks user-interrupts it decrements the mask level. If this returns to zero and the request flag is set, the ULS calls the appropriate handler to service the interrupt.

Second, the tables of sleeping and workahead LWPs in the usched area are written by the ULS and read by the KLS. These tables are read by the KLS only while the VAS is preempted or has yielded. If a VAS is preempted while the ULS is writing the tables, the KLS sees inconsistent data. To prevent this, we need a *VAS preemption masking* mechanism. "Virtual" masking can also be used to implement this mechanism, using a *preemption mask* flag in the usched area and a *preemption request* flag in the ksched area. While the mask is nonzero, the VAS cannot be preempted by another VAS. Upon unmasking preemption, if the ULS finds the request flag set, it calls `yield()`.

Third, several items in the ksched area ($D_{\bar{A}}$, T_{now}, *ready_for_IO*) are written by kernel interrupt handlers (clock, I/O) and read by the ULS. It is possible to do virtual masking of kernel interrupts, but this has the drawback of requiring a system call to service interrupts that occur while kernel interrupts are masked. By exploiting specific properties of these items, simpler solutions are possible. For example, if reading or writing a single word is atomic, then a data structure consisting of a single word (*e.g.*, the *ready_for_IO* flag in an I/O descriptor) requires no synchronization mechanism. For multi-word quantities such as $D_{\bar{A}}$ and T_{now} that are monotonically increasing or decreasing, a consistent value can be obtained by repeatedly reading the quantity until two successive reads result in the same value.

Finally, several items in the usched area (*e.g.*, D_A, T_{next}, *runnable* and *waiting_for_IO*) are read by kernel interrupt handlers and written by the ULS. Again, we can exploit specific properties of these items to achieve simple synchronization mechanisms. Single-word flags require no synchronization if word access is atomic. For multi-word quantities (D_A and T_{next}) the ULS masks preemption during access. If a kernel interrupt handler finds that preemption is masked, it assumes that a multi-word quantity is inconsistent and takes appropriate action. For instance, if T_{next} is inconsistent, the clock interrupt handler delays checking for `INT_TIMER` delivery until the next clock tick; if D_A is inconsistent, the preemption request flag is set.

3.4. Discussion

Split-level scheduling introduces new protection problems: a malicious or incorrect program may keep VAS preemption masked indefinitely, or it may execute indefinitely without changing its deadline. Either of these actions would starve all other VASs. A "watchdog timer" can be used to detect such conditions, and to kill or demote the offending process.

Deadline/workahead scheduling has both "hard" and "soft" variants: the distinction is whether or not processes reserve CPU capacity in advance. In the hard variant, each new LWP specifies its workload (message rate and CPU time per message). The KLS conducts a *schedulability test* to determine whether the workload can be accommodated and if so, with what logical delay bound. This test involves a simulation under worst-case load, and is described in [2]. In the soft variant, no such screening is done, and it is possible for the system to fall behind schedule.

SLS is not restricted to deadline/workahead scheduling; it can be adapted to other policies, such as static priorities or usage-based timesharing policies. The policy dictates the contents of ULS/KLS shared memory; in general, the usched area contains the highest priority among runnable LWPs in the address space, while the ksched area contains the highest priority among runnable LWPs in other address spaces.

4. MEMORY-MAPPED STREAMS

Each real-time LWP in a CM application handles a stream of CM data. The source and sink of each stream are typically I/O devices, and CM data must be moved to or from the kernel address space. A mechanism for this user/kernel IPC has three components:

- **Control and synchronization:** This includes I/O initiation and producer/consumer synchronization.
- **Data location transfer:** If the addresses of data buffers in the user VAS change, they must be transferred from the user to the kernel (if the user determines the buffer addresses) or vice versa.
- **Data transfer:** The actual transfer of data, perhaps by copying or VM remapping.

Traditional user/kernel IPC mechanisms require a user/kernel interaction for one or more of the above components in every I/O operation. For example, the UNIX `read()` system call performs all three components. UNIX asynchronous I/O uses the `read()` system call for data and data location transfer and the `SIGIO` signal and `select()` system call for control and synchronization.

Memory mapped streams (MMS) are a new class of IPC mechanisms for stream-oriented user/kernel IPC[2]. An MMS uses shared memory for control and synchronization. MMSs may use any of a number of techniques for data location transfer; all these use shared memory to hold either the data itself or the data location (with each technique one or more data transfer mechanisms are possible, see Section 4.3). This combination of shared memory mechanisms reduces or eliminates user/kernel interactions in I/O operations.

4.1. Client Interface to Memory-Mapped Streams

The client interface to MMS consists of the following library routines:

```
d = MMS_create(fd, buffer_size, ...);
MMS_read(d, nbytes);
MMS_write(d, nbytes);
```

`MMS_create()` creates a new MMS, returning a descriptor. *Fd* identifies the data source or sink (network connection, disk file, *etc.*); the data direction (read or write) is implicit. *Buffer_size* is the size of the MMS buffer.[3] Additional arguments may be needed for the data transfer structure. `MMS_read()` blocks until `nbytes` of data are available, and `MMS_write()` blocks until `nbytes` of data can be written to the buffer.

[2] The basic technique of MMS (shared-memory synchronization structures) can also be used for user/user IPC. We describe only the user/kernel case here.

[3] Streams in which a storage device sources or sinks data typically have large end-to-end delay bounds (*e.g.*, a second or more), so buffering may be used to increase the system efficiency and responsiveness. Streams that are part of an inter-human conversation or conference have low end-to-end delay bounds (tens of milliseconds) must use smaller buffers. The buffer size may change dynamically; for example, the ACME audio output process must use a small buffer if any of the streams it is currently handling is part of a conversation; otherwise it can use a large buffer.

4.2. Synchronization and I/O Initiation

`MMS_create()` allocates and initializes a *synchronization structure* in an area of memory shared between user and kernel. For concreteness, we discuss the synchronization structure and mechanism for the case when a user LWP (scheduled by a split-level scheduler) reads CM data from an MMS. The synchronization structure contains the following data:

The buffer size.

N_{read}: the number of bytes read so far; this is updated by the LWP.

N_{write}: the number of bytes written so far; this is updated by the kernel. The buffer is empty when $N_{read} = N_{write}$, and full when they differ by the buffer size.

`Active`: a flag, maintained by the kernel. If false, further I/O must be initiated by a request from the user process[4].

B_{wakeup}: if the data level in the MMS is greater than this number, the interrupt handler sets the *ready_for_IO* flag in the MMS descriptor and delivers an `INT_IO_READY` if necessary.

B_{start}: this is set by the kernel. A system call to initiate I/O must be made if the device is not active and the data level falls below this value.

Hysteresis for I/O initiation is controlled by the B_{start} parameter. Hysteresis for process wakeup is effected by appropriately setting B_{wakeup}. The DWS policy for workahead processes also implicitly controls wakeup hysteresis.

The algorithm for `MMS_read()` is as follows:

```
MMS_read(d, n) {
    mask_user_interrupts();
    $B_{wakeup}$ = n;
    waiting_for_IO = TRUE;
    w = $N_{write}$;
    if (w - $N_{read}$ < n)
        IO_wait();
    if ((w - $N_{read}$ < $B_{start}$) && !active)
        initiate_IO();
    waiting_for_IO = FALSE;
    $N_{read}$ += n;
    unmask_user_interrupts();
}
```

[4] A CM I/O device such as a D/A converter is always active; it continually does I/O, periodically generating interrupts when a block of data has been input or output. A file system is generally passive; I/O must be initiated by a system call; this call may trigger a chain of operations via I/O completion interrupts, but eventually another system call is needed to restart I/O. A passive stream, such as a file, can be made active by using a time-based kernel activity (*e.g.*, polling) to restart I/O without intervention from the client. Incoming network connections may be either active or passive depending on the transport protocol used.

This code executes at user level, so I/O interrupts cannot be masked (`mask_user_interrupts()` merely inhibits the delivery of `INT_IO_READY` user interrupts; see Section 3.3). There is a potential race condition if an I/O interrupt occurs between getting N_{write} and calling `IO_wait()`. This race condition is avoided, however, by setting `waiting_for_IO`; if an I/O interrupt occurs during the critical period, it will simply set the `INT_IO_READY` request flag and the descriptor's `ready_for_IO` flag. The ULS will check these flags when it unmasks user interrupts in `IO_wait()`, and will awaken the LWP that called `MMS_read()` if necessary.

The kernel interrupt handler for an n-byte read operation completion does the following:

```
    append data to data transfer structure;
    update data location transfer
        structure if needed;
    $N_{write}$ += n;
    if (waiting_for_IO)
        if ($N_{write}$ - $N_{read}$ > $B_{wakeup}$) {
            ready_for_IO = TRUE;
            if ($C_P$<$T_{now}$ and $D_P$<$D$)
                deliver INT_IO_READY
                user interrupt to VAS
        }
```

Synchronization is simpler in this case because the LWP cannot preempt the interrupt handler.

4.3. Data and Data Location Transfer

The mechanisms for transferring data location, and the data itself, are largely independent of control and synchronization. Some possibilities are:

- Data is passed in pages of physical memory that are statically shared between kernel and user. Data location is implicit. Data copying may still be necessary: for user writing, the kernel may need a copy of the data (*e.g.* for retransmission) after the page has been reused; for user reading, the client may need to write the data to another MMS.
- Data is passed in a fixed range of virtual pages that are mapped dynamically to physical pages. Data location is implicit, and copying can be avoided in some cases.
- The kernel and user share an array of "message descriptors" that contain pointers to blocks of data. Data may be transferred by remapping, by copying, or by copy-on-write.

The optimal choice of mechanism depends on factors such as remapping cost and message size. The control and synchronization mechanism described earlier may have to be slightly modified in some cases; for example, the N_{read} and N_{write} variables may need to be defined in terms of pages or messages instead of bytes.

5. PERFORMANCE

In this section we show by example how split-level scheduling and memory-mapped streams reduce the number of user/kernel interactions. We then compare the performance of split-level scheduled LWPs and MMSs with other alternatives for scheduling and I/O.

5.1. Example Scenario

To see how split-level scheduling and MMSs together reduce the number of user/kernel interactions, consider the following scenario (see Figure 8). An application (say the ACME server) has two real-time LWPs and one background LWP: a device I/O LWP P_D for audio output, a network I/O LWP P_N reading from a CM connection, and an event-handling LWP P_E. P_D has an MMS for output to the audio output device, which interrupts every 30 ms. This MMS's buffer is small (*e.g.*, because the stream it is handling is part of a low-delay conversation). P_N has an input MMS from its network connection; I/O is passive and the MMS buffer is large (*e.g.*, because the data is coming from a file). The LWPs are scheduled using a split-level scheduler. A typical sequence is as follows.

(1) At time 10 ms P_D completes processing a block of audio data and calls `MMS_write()`, which calls `IO_wait()` since the MMS buffer is full. P_N is now the highest priority runnable LWP, so the ULS switches to it.

(2) P_N repeatedly calls `MMS_read()` to wait for a message, processes the message and calls `time_advance()`. At time 27 ms, `MMS_read()` sees that the MMS buffer is empty, and calls `IO_wait()`. P_E is now the highest priority LWP, so the ULS switches to it.

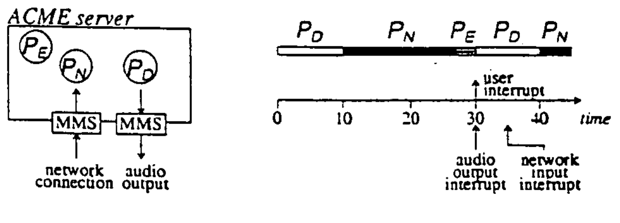

Figure 8: Split-level scheduling and MMSs reduce user/kernel interactions. P_D and P_N do several I/O operations, but there is only a single user/kernel interaction.

(3) At time 30 ms an audio output interrupt occurs. P_D, which was earlier suspended because the MMS buffer was full (see step 1), now becomes critical, so the interrupt handler delivers an `INT_IO_READY`. This causes the ULS to preempt P_E and switch to P_D.

(4) Data arrives for P_N at time 35 ms. Because P_N has worked ahead into the CM stream, $D_{P_N} > D_{P_D}$, and P_N cannot become the highest priority LWP in the VAS. Therefore, the interrupt handler only sets the *ready_for_IO* flag and does not deliver an `INT_IO_READY`.

(5) At time 40 ms, P_D completes the message and calls `MMS_write()`, which calls `IO_wait()` (see step 1 above). From the *ready_for_IO* flag in P_N's MMS descriptor, the ULS finds that the LWP is runnable and also the highest priority LWP. So, the ULS switches to P_N.

In this scenario, the only user/kernel interaction is the user interrupt at time 30 ms. No system calls for I/O or scheduling are needed. An `INT_IO_READY` at time 35 ms is also eliminated.

5.2. Performance Evaluation

In this section we compare the following alternatives for structuring CM applications:

(1) Split-level scheduled LWPs (SLS-LWPs) using MMSs for I/O.

(2) Threads using separate system calls for scheduling and I/O.

(3) LWPs without split-level scheduling (*pure LWPs*) using UNIX asynchronous, non-blocking I/O. In this case, if an LWP does not find data available when it does a non-blocking `read()`, it has to wait for a signal and then do a `select()` before calling `read()` again.

We have implemented prototypes of split-level scheduling and memory-mapped streams, and measured the CPU times of their basic operations on a DECstation 3100 (a 14 MIPS machine representative of current RISC workstations). For the other approaches, we measured scheduling and I/O synchronization costs on a DECstation 3100 running Mach 2.5.

Consider a thread or LWP that reads a CM message from an I/O descriptor, processes the message, and then changes its deadline to that of the next message in the stream. Table 1 shows the total scheduling and I/O synchronization overhead per message for various scenarios.

Scheduling and I/O scenarios	Overhead (in μs)
SLS-LWP reads message from MMS and then calls *time_advance()*	17
SLS-LWP calls *IO_wait()* (because MMS was empty) and is scheduled by *INT_IO_READY*	67
SLS-LWP calls *timed_sleep()* and is awakened by *INT_TIMER*	132
Thread does a system call to read the next message and another to change its deadline	145
Pure LWP does a system call to read the next message and another to change the process deadline	129
Pure LWP does a non-blocking *read()*, later receives a signal, does a *select()* and *read()* to get the message, and finally a system call to change the process deadline	384

Table 1: Scheduling and I/O synchronization overheads per message for different scenarios.

The times shown in Table 1 are a significant fraction of typical CM message processing costs. For instance, scaling a 2K block of 8-bit audio samples takes 1.0 ms, and mixing two 2K blocks takes 1.1 ms. Thus, the scheduling and I/O synchronization overhead using threads and pure LWPs ranges from about 15-25% of the total message processing time.

Figure 9 shows scheduling and I/O synchronization overhead as a function of message rate for a particular workload: an ACME server simultaneously outputting one video stream and two audio streams and inputting an audio stream and distributing it on two CM connections. Thus, the server has five (workahead) network I/O processes and three (periodic) device I/O processes.

Threads incur 4 times the overhead of SLS-LWPs at all message rates; pure-LWPs are 6 times as expensive as SLS-LWPs. Pure LWPs and threads incur CPU overheads of 33% and 23% at 200 messages per second. This message rate is realistic for some low-delay applications which need an end-to-end delay on the order of 10 ms (200 message/sec represents a packetization delay alone of 5ms). Moreover, such a high message rate may also be achieved instantaneously by moderate-delay applications when they are working ahead.

6. RELATED WORK

The work described in this paper is related to several directions of current OS research.

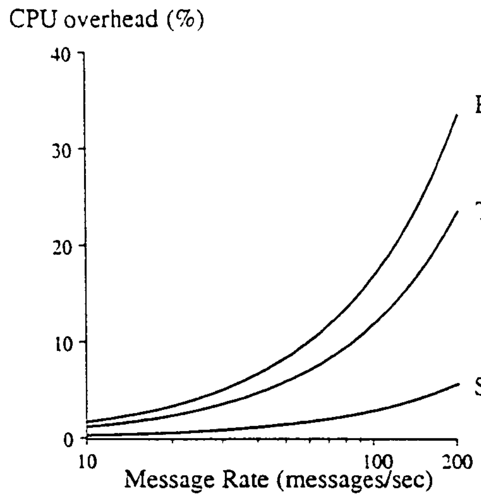

Figure 9: Scheduling and I/O synchronization overhead for an ACME server as a function of message rate. The server workload consists of 5 (workahead) network I/O processes and 3 (non-workahead) device I/O processes.

- **User-level functionality.** Modern operating systems such as Amoeba, Chorus, and Mach shift functionality from kernel to user level to improve software structure. In contrast, our work shifts functionality to user level to increase performance.
- **Asynchronous communication.** Most existing operating systems use request/reply communication; examples include UNIX-type system calls, RPC, and object invocation. This paradigm is not well-suited to continuous media (more generally, it may not be well-suited to future distributed systems in which speed-of-light delays dominate throughput limits). MMSs provide efficient local asynchronous communication. Example of related work include the asynchronous RPC proposed by Gifford [8] and the dataflow model of Synthesis [10].
- **Efficient local data transfer.** In UNIX-type systems, I/O and IPC performance is limited by the overhead of data copying. Systems such as Mach, DASH and Topaz have attacked this problem using techniques such as VM remapping and shared memory [11,12,14]. The MMS mechanism is complementary to this work; it attacks the overhead of control rather than data movement.

SLS is closely related to recent work on multiprocessor operating system support for parallelism, including the Psyche multiprocessor system [13] and scheduler activations [3]. In Psyche, ULSs schedule LWPs on kernel-supported threads. The kernel notifies the user VAS of events, such as blocking cross-domain invocations, that affect the ULS. User/kernel shared memory is used to efficiently communicate LWP identifiers and to request timer user-interrupts. A *scheduler activation* is a thread-like execution context in which an LWP can run. As in Psyche, user-interrupts notify user VASs of scheduling related events. Scheduler activations do not use shared memory to communicate scheduling information between ULSs and the kernel.

In these two approaches, however, the kernel and the ULS scheduling policies are independent. As a result, the approaches cannot correctly prioritize LWPs across threads that may be running in different address spaces that are contending for the the same processor. They also cannot exploit policy-specific information (*e.g.*, LWP priorities) to reduce user-kernel interactions.

For CM applications, the CPU scheduling approach of Synthesis [10] represents an alternative to split-level scheduling. The Synthesis model is based on a rate-control feedback. Processes make no calls to indicate their temporal progress; instead, the kernel adjusts time-slice quanta based on queue lengths. This approach is well-suited to some situations (*e.g.*, audio DSP with little slack CPU time).

The deadline/workahead scheduling policy is derived from the earliest-deadline-first policy [9], but differs in its allowance for workahead.

In Symunix II [7], parallel applications are implemented as a collection of UNIX processes communicating through shared memory. Processes use virtual preemption masking while holding short-duration busy-waiting locks and virtual signal masking while updating shared memory. Unlike virtual user-interrupt masking, virtual signal masking requires a new system call to handle pending signals after unmasking interrupts.

Like MMSs, DEMOS links [5] can have associated shared memory to transfer data between address spaces. However, this approach does not reduce control overhead; synchronization is necessary after each transfer.

MMSs differ from memory-mapped files in several ways. A CM stream may be larger than a VAS, and an MMS need not contain the entire CM stream; since CM streams are accessed sequentially, a small circular buffer suffices. MMSs avoid the overhead of page faults. Also, since data is "released" explicitly, page replacement algorithms are not needed.

Finally, the URPC mechanism developed by Bershad [6] uses shared memory to reduce kernel interaction in local client/server IPC on shared-memory multiprocessors. This is similar in spirit to MMS, though the setting is different.

7. CONCLUSION

Existing operating systems incorporate design principles that are contrary to the needs of applications directly handling real-time streams of continuous media data (digital audio and video):

- The *request/reply* paradigm (the basis of centralized systems as well as RPC-based and object-oriented distributed systems) is non-optimal for stream-oriented CM data.
- Assumptions about temporal locality and delay tolerance of data accesses leads to the use of caching and buffering, which are often inappropriate for CM data.
- Scheduling policies in current systems have the goals of fairness, maximum system throughput, and fast interactive response. CM applications have real-time requirements that may conflict with these goals.

Starting with the goal of supporting CM applications, we have developed two interrelated mechanisms, *split-level scheduling* and *memory-mapped streams*, for scheduling and IPC. We have described their use in a typical CM application (the ACME server) and have compared their performance with that of the analogous mechanisms in UNIX. They improve performance by reducing the number of user/kernel interactions.

Split-level scheduling is most effective when switches between LWPs within a VAS are more frequent than switches between VASs. This is typical of CM systems when there is at most one VAS with low-delay processes (such as ACME's device I/O processes) that require CPU time at frequent intervals. To best exploit split-level scheduling, the I/O server should be the only application run on the workstation. CM playback and record applications have only high-delay processes; hence compute servers and data servers may run multiple applications of this type and still benefit from split-level scheduling.

These mechanisms are applicable for purposes other than CM. For example, memory-mapped streams could be used for access to a sequential disk file or a network stream connection. Process control applications (*e.g.*, [1]) have scheduling requirements similar to those of CM. Split-level scheduling could be used with a time-slicing policy for a situation where a VAS contains both interactive and background processes. More generally, the mechanisms may be useful in any situation where the rate of I/O and scheduling operations, and the cost of user/kernel interactions, are high.

ACKNOWLEDGEMENTS

Eric Barr and Steve Lucco were involved in early discussions of this work. Puneet Kumar at Carnegie-Mellon helped with some of the experiments. We are grateful to Mike Powell and the referees for their constructive comments.

REFERENCES

1. L. S. Alger and J. H. Lala, "A Real Time Operating System For a Nuclear Power Plant Computer", *Proceedings of the 1986 IEEE Real-time Systems Symposium*, 1986, 244-248.
2. D. P. Anderson, "Meta-Scheduling for Distributed Continuous Media", UC Berkeley, EECS Dept., Technical Report No. UCB/CSD 90/599, Oct. 1990.
3. T. E. Anderson, B. E. Bershad, E. D. Lazowska and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", Technical Report No. 90-04-02, Dept. of Computer Science and Engineering, Univ. of Washington, April 1990.
4. D. P. Anderson, R. Govindan and G. Homsy, "Abstractions for Continuous Media in a Network Window System", *International Conference on Multimedia Information Systems*, Singapore, Jan. 1991.
5. F. Baskett, J. H. Howard and J. T. Montague, "Task Communication in DEMOS", *Proc. of the 6th ACM Symp. on Operating System Prin.*, West Lafayette, Indiana, Nov. 16-18, 1977, 23-31.
6. B. Bershad, "High-Performance Cross-Address Space Communication", Technical Report No. 90-06-02, Dept. of Computer Science and Engineering, Univ. of Washington, June 1990.
7. J. Edler, J. Lipkis and E. Schonberg, "Process Management For Highly Parallel UNIX Systems", *Proc. of USENIX Workshop on Unix and Supercomputers*, September 1988, 1-17.
8. D. K. Gifford and N. Glasser, "Remote Pipes and Procedures for Efficient Distributed Computation", *Trans. Computer Systems 6*, 3 (Aug. 1988), 258-283.
9. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *J. ACM 20*, 1 (1973), 47-61.

10. C. Pu, H. Massalin and J. Ioannidis, "The Synthesis Kernel", *Computing Systems 1*, 1 , 11-32.

11. R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Trans. on Computers*, Aug. 1988, 896-908.

12. M. Schroeder and M. Burrows, "Performance of Firefly RPC", *Trans. Computer Systems 8*, 1 (Nov. 1990), 1-17.

13. M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos and N. G. Smithline, "Implementation Issues For the Psyche Multiprocessor Operating System", *Computing Systems 3, 1* (Winter 1990), 101-137.

14. S. Tzou and D. P. Anderson, "The Performance of Message-Passing Using Restricted Virtual Memory Remapping", *Software – Practice & Experience 21*, 3 (March 1991).